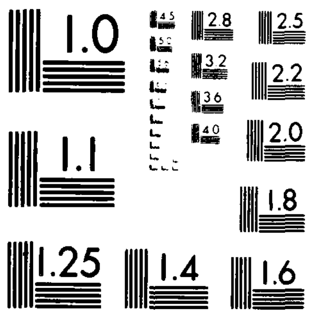
AD-A086 292   NORTHWESTERN UNIV EVANSTON IL DEPT OF ELECTRICAL ENG--ETC F/6 9/2
SELF-METRIC SOFTWARE. VOLUME III. A HANDBOOK. PART II. PERFORMA--ETC(U)
APR 80   S S YAU, J S COLLOFELLO   F30602-76-C-0397
UNCLASSIFIED   RADC-TR-80-138-VOL-3   NL

1 OF 1
AD
A086292

END
DATE
FILMED
8-80
DTIC


1.0
4.5
5.0
5.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

LEVEL III

5c

A082 2nd (12)

RADC-TR-80-138 Vol III (of three)
Final Technical Report
April 1980

SELF-METRIC SOFTWARE
A Handbook: Part II, Performance Ripple Effect Analysis

Northwestern University

Stephen S. Yau
James S. Collofello

ADA086292

PT 1 - A086291
see A086290 also

[illegible]

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-80-138, Vol III (of three) has been reviewed and is approved for publication.

APPROVED: [illegible]

ROCCO F. IUORNO
Project Engineer

APPROVED: [illegible]

WENDALL C. BAUMAN, Col, USAF
Chief, Information Sciences Division

Accession For	
NTIS GRA&I	✓
DDC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

FOR THE COMMANDER: [illegible]

JOHN P. HUSS
Acting Chief, Plans Office

[illegible]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-80-138, Vol III (of three)	2. GOVT ACCESSION NO. AD-A086 292	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SELF-METRIC SOFTWARE. Volume III. A Handbook, Part II, Performance Ripple Effect Analysis.		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Aug 76 - Jan 80
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Stephen S. Yau, James S. Collofello		8. CONTRACT OR GRANT NUMBER(s) F30602-76-C-0397
9. PERFORMING ORGANIZATION NAME AND ADDRESS Northwestern University, Department of Electrical Engineering & Computer Science, Evanston IL 60201		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55810278
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		12. REPORT DATE April 1980
		13. NUMBER OF PAGES 47
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

9 Final [illegible]

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Rocco F. Iuorno (ISIS)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software maintenance process, performance ripple effect analysis, mechanisms for modification propagation, technique, handbook, lexical analysis and tracing, propagation mechanisms, performance attributes and critical sections.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This handbook consists of two parts on ripple effect analysis for large-scale software maintenance. In Part I, a ripple effect analysis technique for software maintenance from the logical or functional perspective is presented. In this volume, the Part II of the handbook, a ripple effect analysis technique for software maintenance from the performance perspective is presented. The purpose of this handbook is to present ripple effect analysis techniques to assist software maintenance person-

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED (Cont'd)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

411 [illegible]

Item 20 (Cont'd)

nel to do a better job in large-scale software maintenance. The material presented in this handbook is organized in three levels. At the first level, the software maintenance process is described and the need for effective ripple effect analysis techniques for large-scale software maintenance is given. The capabilities and restrictions of the performance ripple effect analysis technique, as well as how this techniques is interfaced with the user, are presented. At the second level, the performance ripple effect analysis technique is outlined in two phases: the lexical analysis phase and the tracing phase. At the third level, the steps of the performance ripple effect analysis technique is not presented in this handbook, but contained in other reports. Finally, the integration of the processing steps of the performance ripple effect analysis and the logical ripple effect analysis is discussed.

Table of Contents

Table of Contents

List of Figures

List of Tables

1.0 Introduction

The high cost of software maintenance and the urgent need for techniques to improve the software maintenance process were discussed in Part I of this handbook. As we have discussed before, the complexity of software maintenance is primarily due to the ripple effect of software modification, and a logical ripple effect analysis technique has been presented in Part I of the handbook [1]. Since a large-scale program usually has both functional and performance requirements, the ripple effect of program modifications must be analyzed from both a functional and a performance point of view. Thus, in this part of the handbook, a performance ripple effect analysis technique, which should be used in conjunction with the logical ripple effect analysis technique will be presented.

The perspective of performance ripple effect analysis technique, as well as the logical ripple effect analysis technique in the software maintenance process is shown in Figure 1. Figure 2 expands the box of performance ripple effect analysis in Figure 1 to illustrate the inputs and outputs of the performance ripple effect analysis technique. The outputs of this technique in conjunction with that of the logical ripple effect analysis technique can help maintenance personnel understand the scope of effect of their changes on the program. The results can also aid the maintenance personnel in determining which parts of the program must be examined for consistency or possible optimization to improve performance. The net results of applying the logical and performance ripple effect analysis techniques are:

* Smoother implementation of program modifications
* Reduction of program errors introduced in the program due to modifications
* Reduction of program structure degradation as a consequence of program modification due to an increased understanding of the implications of the modification
* Decrease of the growth rate of program complexity due to program modification
* Extension of overall program's operating life

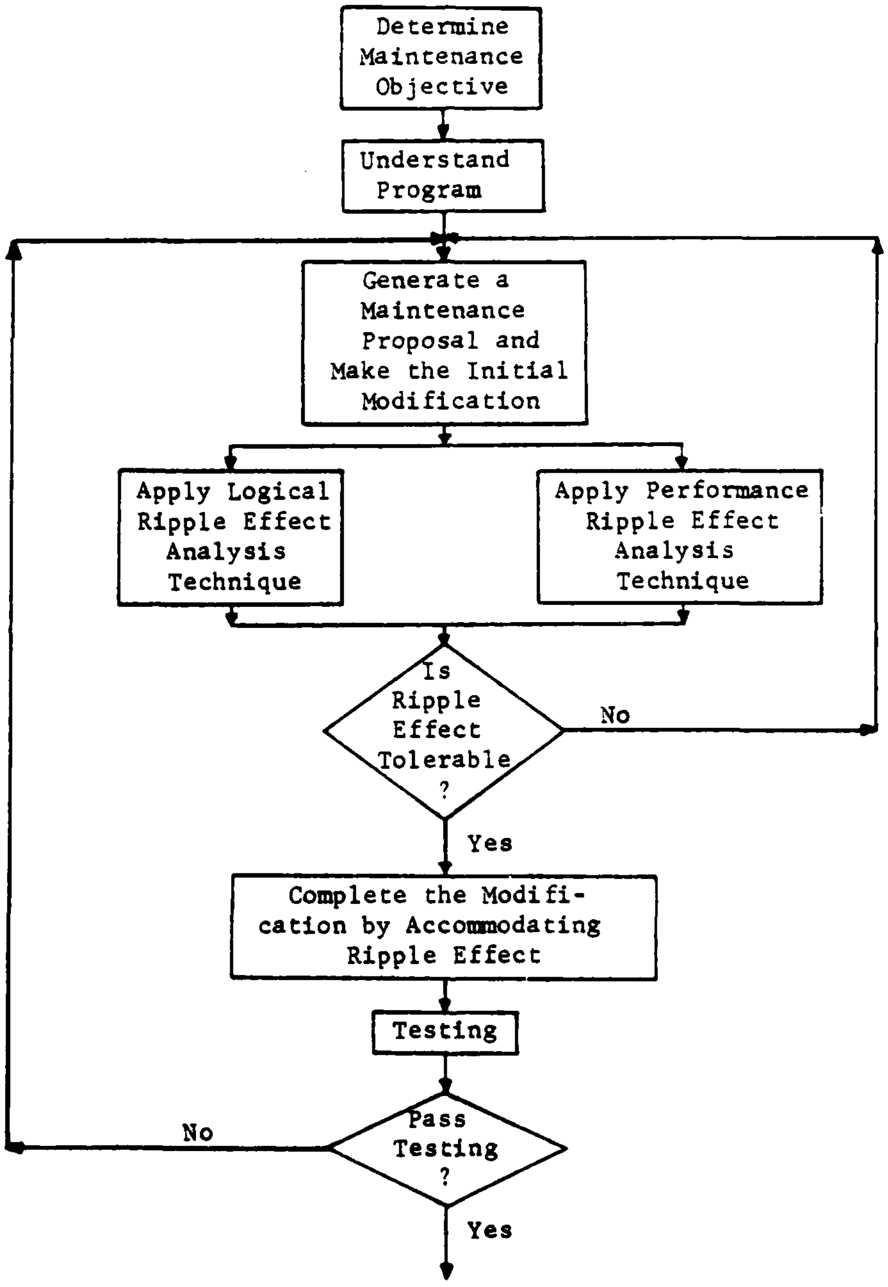

Determine Maintenance Objective
Understand Program
Generate a Maintenance Proposal and Make the Initial Modification
Apply Logical Ripple Effect Analysis Technique
Apply Performance Ripple Effect Analysis Technique
Is Ripple Effect Tolerable ?
No
Yes
Complete the Modification by Accommodating Ripple Effect
Testing
Pass Testing ?
No
Yes

Figure 1. A software maintenance process with the ripple effect analysis techniques

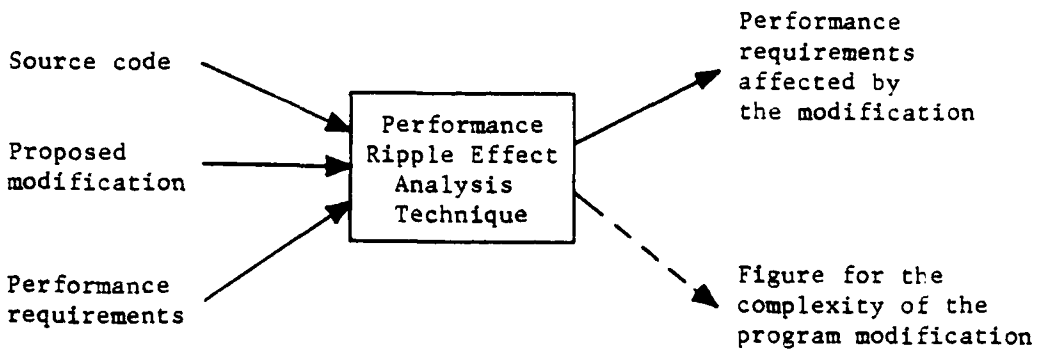

Source code
Proposed modification
Performance requirements
Performance Ripple Effect Analysis Technique
Performance requirements affected by the modification
Figure for the complexity of the program modification

Figure 2. Illustration of the inputs and outputs of the performance ripple effect analysis technique.

Another significant product of the logical and performance ripple effect analysis techniques is the computation of the complexity of a proposed program modification. One such figure has been proposed which reflects the amount of work involved in performing maintenance and thus provides a standard on which comparisons of modifications can be made [2]. However, further research is required for estimating such a figure.

The objective of Part II of this handbook is to describe the performance ripple effect analysis technique in a clear and concise manner. Section 2 describes the capabilities and restrictions of the technique. Section 3 presents the user level interface to this technique as it is perceived to be when fully implemented. Section 4 outlines the required processing necessary to accomplish the functions described in Section 3. Section 5 deals with a description of each of the processing steps. Section 6 integrates the required processing steps of the performance ripple effect analysis technique with that of the logical ripple effect analysis technique. This handbook does not contain implementation details or verification of the algorithms described. Further information of this type is discussed in other more detailed reports [3-5].

2.0 Capabilities and Restrictions of the Performance Ripple Effect Analysis Technique

The performance ripple effect analysis technique described in this part of the handbook is language independent and applicable to existing programs as well as newly implemented programs incorporating state-of-the-art design techniques. The technique does not provide maintenance personnel with proposals for modifying the program. Instead, the technique is applied after the maintenance personnel have generated a number of possible maintenance proposals.

The current version of the performance ripple effect analysis technique makes the following assumptions about the program to be analyzed.

1. The performance requirements for the program are testable and stated in terms of flows through the program. If the performance requirements are stated in a machine readable requirements statement language, then the technique can be fully automated.

2. Parallel execution can only exist at the module level. This is an assumption for simplifying the technique and can easily be eliminated at a cost of using more complex algorithms.
3. The ability of a module to execute in parallel with another module is reflected in the software implementation or its accompanying documentation.
4. Execution priorities are reflected in the software implementation or its accompanying documentation.

3.0 User Interface

The success of any software technique depends on its ease of use. The technique must be simple to understand and apply to the problem. This implies a high degree of automation in which the user interfaces with the technique at a very high level, and the technique is transparent to the user on how it operates.

The performance ripple effect analysis technique has been developed with these objectives. When the technique is fully automated, it is very easily applied to the problem. Although the technique is very sophisticated, the maintenance personnel applying the technique need only be concerned with its output. The performance ripple effect analysis technique is applied in the following three simple steps which are illustrated in Figure 3.

Step 1: Maintenance personnel utilize a change management system (CMS) to modify the program. The CMS consists of a text editor and a data base. The CMS records all of the changes in the program automatically in the data base. Thus, a record of the maintenance activity is created without special assistance from the maintenance personnel.
Step 2: After the modification of the program is complete, the maintenance personnel execute the lexical analysis package of the performance ripple effect analysis technique.
Step 3: Upon completion of the lexical analysis step, the maintenance personnel execute the tracing package which utilizes the data base of program changes created by the CMS and maps these changes into the characterization of the program created by the lexical analysis step. It then traces performance ripple effect throughout the program. The output of the tracing package is a listing of the performance requirements affected by performance ripple effect.

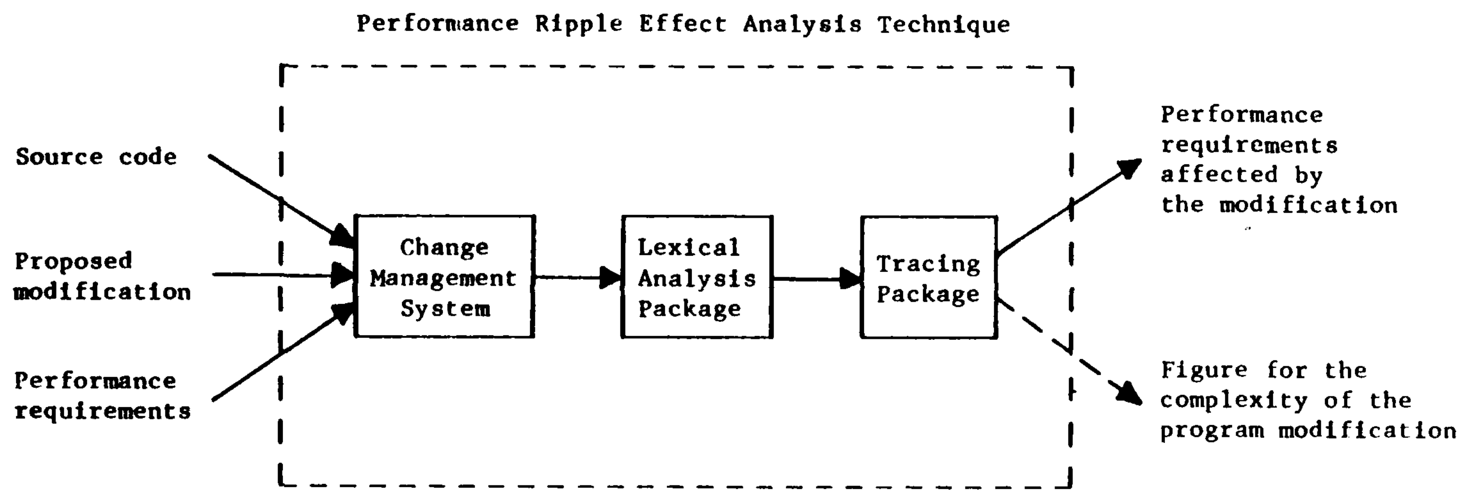

Performance Ripple Effect Analysis Technique
Source code
Proposed modification
Performance requirements
Change Management System
Lexical Analysis Package
Tracing Package
Performance requirements affected by the modification
Figure for the complexity of the program modification

Figure 3. User interface level of the performance ripple effect analysis technique

The performance ripple effect analysis technique, thus, provides maintenance personnel with valuable information about the maintenance activity without interfering with the maintenance process itself or requiring additional input from the maintenance personnel.

4.0 Outline of Performance Ripple Effect Analysis Technique

In this section, we will outline the processing steps involved with the lexical analysis and tracing phases of the performance ripple effect technique.

4.1 Lexical Analysis Phase

The first phase of the performance ripple effect analysis technique is the lexical analysis phase. In this phase, the program is analyzed with respect to the proposed modification and a characterization of the program is compiled and saved in a data base. The characterization of the program contains information necessary for tracing performance ripple effect. A description of the performance information needed for this characterization will now be presented.

4.1.1 Performance Characterization of the Program

The ability to trace performance ripple effect in a program requires the identification of modules whose performance may change as a consequence of software modifications. This is a complex task because performance dependencies often exist among modules which are otherwise functionally and logically independent. A performance dependency relationship (PDR) is defined to exist from module A to module B if and only if a change in module A can have an effect on the performance of module B. A performance interdependency relationship (PIR) is defined to exist between two modules A and B if a PDR exists from module A to module B and a PDR exists from module B to module A.

It is obvious that when a logical error is discovered in the software, this error can affect other modules. Analogously, when a performance change is introduced, the scope of effect of the change can be determined by examining the performance dependency relationships in existence in the program. These performance dependency relationships are determined by identifying the mechanisms by which performance changes are propagated in a program. We have identified eight mechanisms which may exist in large-scale programs by which changes in performance are propagated throughout the program [3,4]. The identification of which mechanisms exist in the program and which performance dependency relationships exist between the modules via these mechanisms are important components of the performance characterization of the program produced during lexical analysis.

Performance attributes of a program are defined as attributes corresponding to measurements of key aspects of the execution of the program. There exists a distinct relationship between performance attributes and the eight mechanisms for the propagation of performance changes. The eight mechanisms operate as links between performance attributes such that a change in a performance attribute of one module can affect a performance attribute in another module via one of the eight mechanisms. Fourteen performance attributes for large-scale programs have been identified [4]. The identification of these performance attributes is also an important component of the performance characterization of the program produced during lexical analysis.

There is also a relationship between the performance attributes and the performance requirements of the program. Performance requirements can be decomposed qualitatively into performance attributes which contribute to the preservation or violation of the performance requirements. This decomposition depends on the method utilized in specifying the performance requirements. Guidelines for the decomposition of performance requirements stated at a level, such as the R-Net level, have been developed [3,4]. As requirement statement languages [6] continue to develop, it appears feasible that the decomposition of performance requirements into performance attributes may be accomplished automatically. Thus, a set of performance attributes can be associated with each performance requirement such that if a performance attribute in the set is affected by a modification, then the performance requirement associated with this set is also affected. This decomposition of performance requirements into the program's performance attributes is an important component of the performance characterization.

Since the performance attributes of a program correspond to measurements of key aspects of the execution of the program, they can be affected during the maintenance process by modification to the program. A critical section of a program can be associated with each performance attribute such that if this critical section is modified, the corresponding performance attribute may be affected. Seven types of critical sections for large-scale programs have identified [4]. The identification of these critical sections is an important component of the performance characterization of the program produced during lexical analysis.

When a performance attribute is affected by a modification, the other performance attributes affected by the change must be identified. It is not, however, always possible to identify exactly which performance attributes are affected as a consequence of the performance ripple effect. When a particular performance attribute is affected by a modification, it is possible to identify the critical sections of code which may experience the performance change. Corresponding to each critical section, there may be several performance attributes. The concept of a virtual performance attribute is introduced to represent this change in performance of a critical section.

A virtual performance attribute is defined to represent a change in performance of a critical section which is a consequence of affecting some performance attribute in the program. If a performance attribute is involved in a performance dependency relationship with a virtual performance attribute, it means that a change in the performance attribute will affect the virtual performance attribute, i.e. the performance of some software section. Corresponding to the virtual performance attribute, there may be many performance attributes. Five virtual performance attributes for large-scale programs have been identified [4]. The identification of these virtual performance attributes is an important component of the performance characterization of the program produced during lexical analysis.

4.1.2 Outline of the Procedure to Perform Lexical Analysis

The processing steps involved with lexical analysis can be summarized as follows:

Step 1: Identify all of the mechanisms for the propagation of performance changes and the corresponding performance dependency relationships in the program.

Step 2: Identify all of the critical sections in the program.

Step 3: Identify all of the performance attributes in the program.

Step 4: Identify all of the virtual performance attributes in the program.

Step 5: Decompose the performance requirements for the program into the performance attributes which contribute to the preservation or violation of the performance requirements.

4.2 Tracing Phase

The second phase of the performance ripple effect analysis maintenance technique consists of tracing the performance changes, i.e. the performance ripple effect which occurs as a consequence of the maintenance changes. The input to the technique in this phase includes all of the information about the program collected and stored in a data base during the lexical analysis phase.

4.2.1 Performance Ripple Effect Tracing

Tracing performance ripple effect is a complex task, and is accomplished by analyzing the performance attributes which are affected by the modification. When a particular performance attribute is affected, it may affect other performance attributes in the program. These affected performance attributes may then in turn affect other performance attributes which accounts for the ripple effect.

This ripple effect of performance changes can be analyzed through the use of performance dependency relationship rules. These rules represent the conditions by which a change in a performance attribute can affect another performance attribute or a virtual performance attribute. A complete set of performance dependency relationship rules for tracing ripple effect has been developed [4].

4.2.2 Outline of the Procedure to Perform Ripple Effect Tracing

In this section, the processing steps for tracing performance ripple effect will be presented in the required order.

Step 1: Utilizing the change management system data base and the characterization of the program produced during lexical analysis, identify the set of blocks involved in the change for each module in the program.

Step 2: Based on the blocks involved in the change identified in the last step and the characterization of the program produced during lexical analysis, identify all of the critical sections affected by the maintenance activity.

Step 3: For each of the critical sections affected by the modification, determine the corresponding performance attributes which may be affected if the critical section is modified. (The correspondence between performance attributes and critical sections was generated during lexical analysis.)
Step 4: Trace the performance ripple effect among the performance attributes utilizing the performance dependency relationship rules in order to identify all of the performance attributes affected by the modification.
Step 5: Identify the performance requirements which are affected by a change in any of the performance attributes involved directly with the modification or through its ripple effect. These performance requirements can be identified by the traceability of the decomposition of the performance requirements into the performance attributes performed during lexical analysis.

5.0 Description of Each Step of the Technique

In this section, a description of each of the steps involved in the lexical analysis and tracing phases of the performance ripple effect analysis will be provided. The description will be informal and concise. The processing steps will be described at a level which is language independent. Informal algorithms and approaches used in these steps will also be presented, but the actual implementation is language dependent and hence omitted.

5.1 Description of Lexical Analysis Steps

In this section, a description will be presented for each of the lexical analysis steps outlined in Section 4. Section 5.2 will contain a description of each of the steps in the tracing phase which has also been outlined in Section 4.

5.1.1 Lexical Analysis Step 1

In this step, all of the mechanisms for the propagation of performance changes and the corresponding performance dependency relationships in the program are identified. These performance dependency relationships can be saved in a data base for later use in other processing steps. In the following sections, a description of how to identify each of the mechanisms will be presented.

5.1.1.1 Identification of Parallel Execution Mechanism

Software modifications to a module can destroy the ability of the module to execute in parallel with other modules and lead to major changes in performance due to execution delays and contention for resources previously alleviated through the parallel execution. The identification of the existence of the parallel execution mechanism in a program and the modules in the program affected by this mechanism requires identification of which modules in the program can be executed in parallel. Parallel execution considered here is restricted to the module level for simplicity purpose. The determination of which modules can be executed in parallel is a decision made during the design phase of the program. This decision must be reflected in either the software implementation or its accompanying documentation.

Parallel execution can be illustrated using various control flow graph techniques. The program's control flow path can then be analyzed to identify which modules were designed to be executed in parallel. A PIR is then defined to exist between each pair of modules executable in parallel. If module A and module B are executable in parallel, PIRs are also defined to exist between each module in the set of modules directly or indirectly invocable from module A, including module A, and each module in the set of modules directly or indirectly invocable from module B, including module B. A module X is directly or indirectly invocable from a module Y if module X is in the subtree with root Y of the module invocation graph of the program. For example, consider the invocation graph for the sample program in Figure 4 and assume that modules 4 and 6 are executable in parallel. The set of modules directly or indirectly invocable from module 4 is {7,11,12}. The set of modules directly or indirectly invocable from module 6 is {9,10,14}. Thus, PIRs via the parallel execution mechanism of modules 4 and 6 can be identified for the following pairs of modules: (4,6), (4,9), (4,10), (4,14), (6,7), (6,11), and (6,12).

Algorithms can easily be developed for systematically analyzing the control flow graph structure of large scale programs to identify the modules affected by the parallel execution mechanism. One such algorithm developed for analyzing an R-Net control flow structure is formally described elsewhere [4].

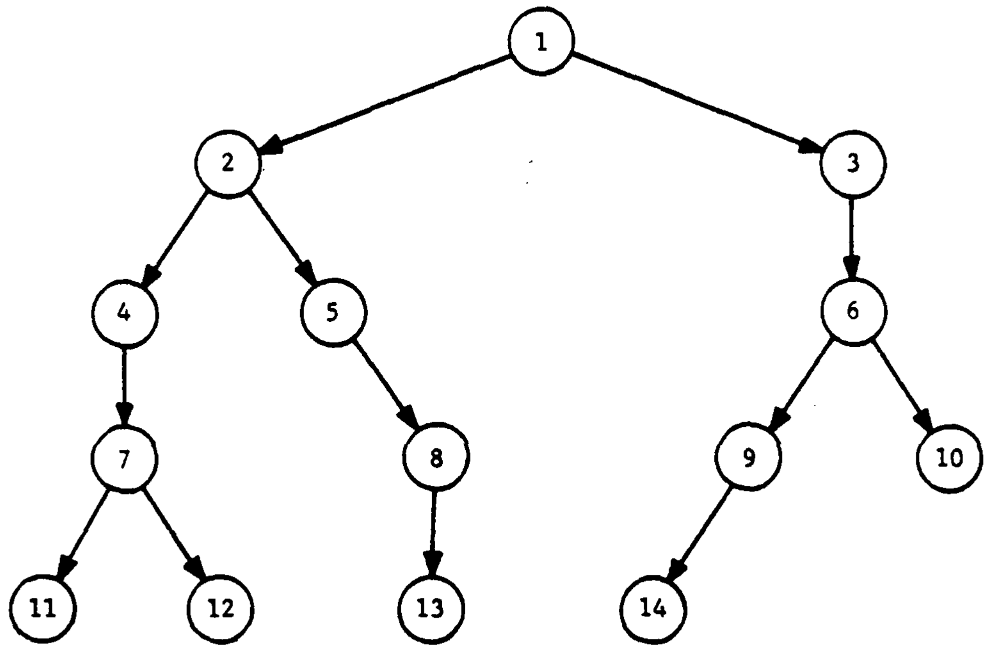


Figure 4. The invocation graph of a sample program as an example of identification of performance interdependency relationships via the parallel execution mechanism.

5.1.1.2 Identification of Shared Resources Mechanism

Software modifications to modules sharing resources affecting the time when the modules request and release common resources can result in performance degradation by the other modules whose execution is affected by the denial of requested resources which are currently dedicated to other modules. The identification of the existence of the shared resources mechanism in a program and the modules in the program affected by this mechanism requires the identification of modules sharing common resources and executable in parallel. Sets of modules executable in parallel can be identified by the parallel execution mechanism. Requests for resources can also be identified during compilation. Resources are defined in the standard terminology and include such things as programs, files, devices, etc. Thus, for each resource, a set of modules utilizing the resource can be identified by a static analysis of the code. For each resource and the set of modules utilizing the resource, a subset of these modules executable in parallel can be identified. This subset of modules is utilizing the resource at the same time. If the number of modules in this subset is greater than the number of resources of the same type available, then the modules are competing for the resources. A PIR is then defined to exist between each pair of modules which are competing for the same resource via the shared resources mechanism.

5.1.1.3 Identification of Interprocess Communication Mechanism

Software modifications to a module affecting the time when it transmits a message to a waiting module can affect the performance of the module designated to receive the message. Interprocess communication can be identified in the software when synchronization primitives such as P and V operators or WAIT and POST macros are utilized. It is then possible to perform a static analysis of the program to identify the modules involved in the communication. A PDR is then defined to exist between the module sending the message and the module designated to receive the message. If module A is transmitting a message to module B, PDRs are also defined to exist between each module in the set of modules which directly or indirectly invokes module A and module B. For example, consider the invocation graph for the sample program in Figure 5 and assume that module 4 is transmitting a message to module 6. The set of modules which directly or indirectly invokes module 4 is {1,2}. Thus, PDRs via the

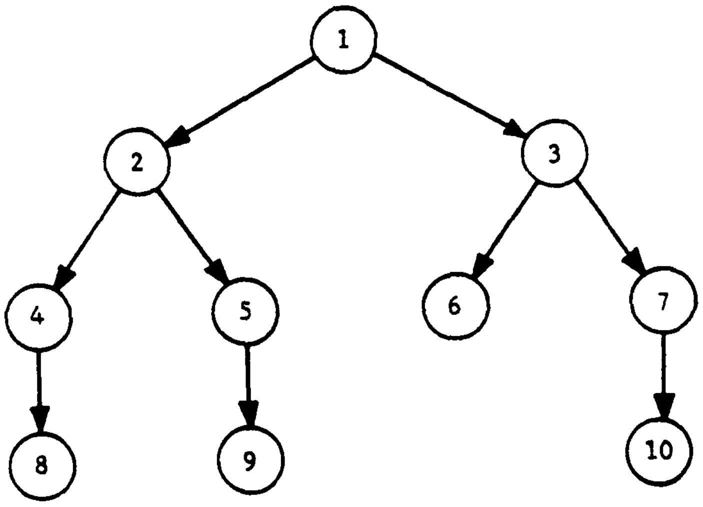


Figure 5. The invocation graph of a sample program as an example of identification of performance dependency relationships via the interprocess communication mechanism.

interprocess communication mechanism can be defined for the following pairs of modules, where the PDR exists from the first module in the pair to the second: (4,6), (1,6), (2,6).

5.1.1.4 Identification of the Called Modules Mechanism

Software modifications to a module can affect the performance of every module which directly or indirectly invoke it. For example, a change to a module affecting its execution time will affect the execution time of all the modules which directly or indirectly invoke it. The invocation structure of a program can easily be constructed by a static analysis of the code. One tool already available for performing this task is JAVS [7]. For each module X, a PDR is then defined from each module in the set of modules directly or indirectly invocable from module X to module X. For example, consider the invocation graph for the sample program in Figure 6. PDRs via the called module mechanism can be defined for the following pairs of modules, where the PDR exists from the first module in the pair to the second: (5,2), (5,1), (2,1), (9,6), (9,2), (9,1), (6,1), (6,2), (3,1), (4,1).

5.1.1.5 Identification of the Shared Data Structures Mechanism

Software modifications affecting the contents of shared data structures can affect the storage and retrieval times for entries in the data structure by other modules sharing the data structure. Modifications affecting the quantity of data stored must be analyzed to determine the effect on the performance of the modules utilizing the shared data structure.

The modules manipulating shared data structures can be classified into one or more of the following categories based upon their utilization of the data structure:

* Reference entries only
* Update entries
* Create new entries
* Delete old entries

Shared data structures and the modules manipulating them can easily be identified by a static analysis of the program [4]. Shared data structures can be identified as global constructs or passed parameters. References to

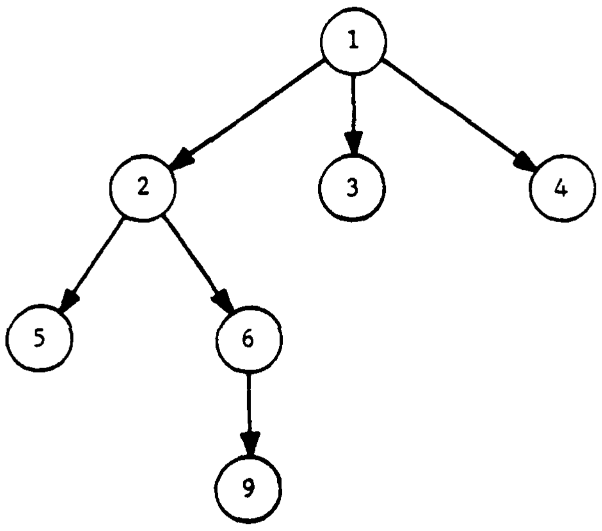


Figure 6. The invocation graph of a sample program as an example of identification of performance dependency relationships via the called modules mechanism.

these shared data structures are also identifiable by a compiler. These references can then be placed in one of the above categories. PDRs via the shared data structures mechanism are then defined to exist between each module in the set of modules which creates or deletes entries in the shared data structure and each module in the set of modules which shares the data structure.

5.1.1.6 Identification of the Sensitivity to the Rate of Input Mechanism

Changes in input rates to a process can lead to saturation and possibly overflow of data structures involved with the processing of the input. The increased input rate can also lead to increased interruptions in processing and possibly performance requirement violations. The identification of the existence of the sensitivity to the rate of input mechanism in a program and the modules in the program affected by this mechanism requires the identification of which modules interface with the environment to handle input interrupts. These modules can be identified by a static analysis of the code. A PDR is then established for these modules such that if the rate of input changes, then a performance change in the modules will result.

5.1.1.7 Identification of the Execution Priorities Mechanism

Software modifications affecting execution priorities in the program can create conflicts such as resource contention that can lead to performance degradation. In addition, if module A has the ability to interrupt the execution of module B, then modifications to module A affecting its execution time can affect the performance of module B since module B must wait until module A completes its execution before module B can complete its execution. Execution priorities are established in the program during the software development phase. These priorities must be reflected in either the software implementation or in the accompanying documentation. Thus, for each module X in the program, we can identify the set of modules for which X has interrupt priority, i.e. the set of modules that can be interrupted by module X. A PDR is then defined to exist from module X to each of the modules in this set. If module X has interrupt priority over module Y, PDRs via the execution priority mechanism are also defined to exist between module X and each module in the set of modules directly or indirectly invocable from module Y.

5.1.1.8 Identification of the Abstraction Mechanism

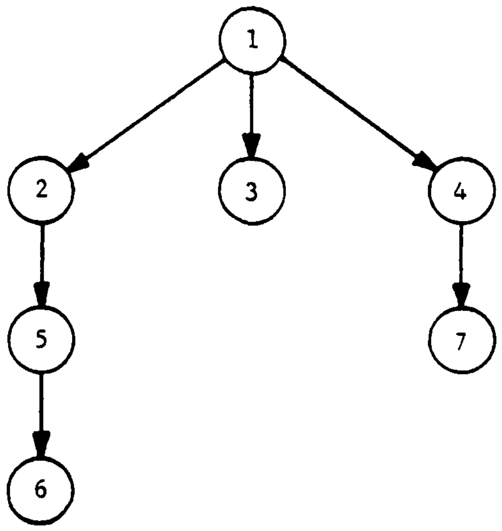
Control and data abstractions are popular design tools that increase the maintainability of a program by hiding design decisions. From a performance perspective, however, these abstractions can decrease the maintainability of a program. This is because a change in the implementation of the abstraction will very likely affect the performance of the abstraction, and hence the performance of all the modules utilizing the abstraction. The utilization of abstractions in a module can be easily identified by static analysis. Abstractions can be recognized in the module as subroutine calls, function calls, and macros. PDRs can then be defined from the implementations of the abstractions to the modules utilizing them. If module A utilizes abstraction X, then PDRs exist from the implementations of all abstractions directly or indirectly utilized by abstraction X to module A. An abstraction Y is directly or indirectly utilized by an abstraction X if X utilizes the abstraction Y or X utilizes an abstraction which directly or indirectly utilizes abstraction Y. For example, consider the invocation graph and static analysis information for the sample program in Figure 7. PDRs via the abstraction mechanism can be identified from the implementations of abstractions A1, A2, and A3 to module 4 and from abstraction A2 to module 5.

5.1.2 Lexical Analysis Step 2

In this section all the critical sections in the program will be identified. The identification of the critical sections utilizes the data base of performance dependency relationships created during the identification of the mechanisms for the propagation of performance changes. The critical sections are identified in terms of program blocks. A critical section data base is then created in such a manner that it is possible to identify which critical sections are associated with a particular program block. There are seven types of critical sections. In the following sections, a description of how to identify each of the critical section types will be presented.

5.1.2.1 Identification of Critical Sections of Type One

A critical section of type one is defined for all modules involved in a PIR via the parallel execution mechanism and consists of all of the blocks in the module. The identification of these critical sections requires two steps.



<u>Summary of Utilization of Abstractions</u>

Module 4 invokes abstraction A1.
Module 5 invokes abstraction A2.

Abstraction A1 utilizes abstraction A3.
Abstraction A3 utilizes abstraction A2.

Figure 7. An example of identification of performance dependency relationships via the abstraction mechanism.

Step 1: Identify all of the modules involved in a PIR via the parallel execution mechanism.

Step 2: For each of these modules, designate all of the blocks in the module to be in a critical section of type one for the module.

5.1.2.2 Identification of Critical Sections of Type Two

A critical section of type two is defined for all modules involved in a PIR via the shared resources mechanism and consists of all of the blocks in the module between its invocation and request for the resource in contention. It may also consist of all of the blocks in the module between its invocation and its call to a module which is involved in a PIR via the shared resources mechanism. The identification of these critical sections requires the following steps:

Step 1: For each module involved in a PIR via the shared resources mechanism, designate the blocks in the module between its invocation and its request for the resource in contention as in a critical section of type two for the module. If the module contains multiple requests for the resource, it will have multiple critical sections.

Step 2: For each module involved in a PIR via the shared resources mechanism, designate the blocks in the module between its invocation and its call to another module involved in a PIR via the shared resources mechanism as in a critical section of type two for the module. If the module contains multiple calls to modules involved in PIRs via the shared resources mechanism, it will have multiple critical sections.

5.1.2.3 Identification of Critical Sections of Type Three

A critical section of type three is defined for all modules involved in a PIR via the shared resources mechanism which request the resource in contention and consists of all of the blocks in the module between its request for the resource in contention and its release of the resource. The identification of these critical sections requires the following steps:

Step 1: Identify all of the modules involved in a PIR via the shared resources mechanism.

Step 2: For each of these modules which directly requests the resource in contention, designate the blocks in the module between its request for the resource in contention and its release of the resource as in a critical section of type three for the module.

5.1.2.4 Identification of Critical Sections of Type Four

A critical section of type four is defined for all dominant modules involved in a PDR via the interprocess communication mechanism and consists of all of the blocks in the module between its invocation and its transmission of the message. A module X is the dominant module involved in a PDR with module Y, if a change in module X can affect module Y via the PDR. The critical section may also consist of all blocks in the dominant module between its invocation and its call to another dominant module involved in a PDR via the interprocess communication mechanism. The identification of these critical sections requires the following steps:

Step 1: For each dominant module involved in a PDR via the interprocess communication mechanism, designate the blocks in the module between its invocation and its transmission of the message as in a critical section of type four for the module. If the module contains multiple transmissions of messages, it will have multiple critical sections.

Step 2: For each dominant module involved in a PDR via the interprocess communication mechanism, designate the blocks in the module between its invocation and its call to another dominant module involved in a PDR via the interprocess communication mechanism as in a critical section of type four for the module. If the module contains multiple calls to dominant modules involved in PDRs via the interprocess communication mechanism, it will have multiple critical sections.

5.1.2.5 Identification of Critical Sections of Type Five

A critical section of type five is defined for all dominant modules involved in a PDR via the called modules mechanism, the execution priorities mechanism or the abstraction mechanism and consists of all the blocks in the module. The identification of these critical sections requires the following steps:

Step 1: Identify all of the dominant modules involved in a PDR via the called modules mechanism, the execution priorities mechanism, or the abstraction mechanism.

Step 2: For each of these modules, designate all of the blocks in the module as in a critical section of type five for the module.

5.1.2.6 Identification of Critical Sections of Type Six

A critical section of type six is defined for all dominant modules involved in a PDR via the called modules mechanism or the abstraction mechanism and consists of all of the blocks in the module between its request for a resource and its release of the resource. The identification of these critical sections requires the following steps:

Step 1: Identify all of the dominant modules involved in a PDR via the called modules mechanism or the abstraction mechanism.

Step 2: For each of these modules, designate the blocks in the module between its request for a resource and its release of the resource as in a critical section of type six for the module. If the module utilizes multiple resources, it will have multiple critical sections of this type.

5.1.2.7 Identification of Critical Sections of Type Seven

A critical section of type seven is defined for all modules containing a dependent iterative structure. A dependent iterative structure is an iterative structure which does not possess a constant number of iterations, i.e. it has a variable number of iterations dependent upon certain program variables. Each dependent iterative structure has its own critical section. The identification of these critical sections requires the following steps:

Step 1: Examine each module and identify the dependent iterative structures in the module.

Step 2: For each dependent iterative structure in each module, designate the set of variables involved in the iterative structure, excluding the index variable as a critical section of type seven for the module.

5.1.3 Lexical Analysis Step 3

In this section all of the performance attributes in the program will be identified. A data base of performance attribute information will be created which associates performance attributes with modules and data structures. Many performance attributes will also be associated with particular critical sections such that if the critical section is modified, the corresponding performance attribute affected can be identified. Each performance attribute will now be defined and a description of how it is identified will be presented.

Performance Attribute 1: The ability of the module to execute in parallel with another module.

This performance attribute is associated with critical sections of type one.

Performance Attribute 2: For each resource in contention, the relative time that the module seizes the resource.

This performance attribute is associated with critical sections of type two.

Performance Attribute 3: For each resource in contention, the relative time that the module releases the resource.

This performance attribute is associated with critical sections of type three.

Performance Attribute 4: The relative time that the module begins execution.

This performance attribute is associated with critical sections of type four.

Performance Attribute 5: The relative time that a module transmits a message to another module.

This performance attribute is associated with critical sections of type four.

Performance Attribute 6: The execution time of the module.

This performance attribute is associated with critical sections of type five.

<u>Performance Attribute 7</u>: For each resource utilized in the module, the resource utilization by the module. Resource utilization is defined as the time that the module possesses the resource.

This performance attribute is associated with each module which utilizes a resource. It is also associated with critical sections of type six.

<u>Performance Attribute 8</u>: For each data structure, the storage and retrieval times for entries in the data structure.

This performance attribute is associated with each data structure in the program.

<u>Performance Attribute 9</u>: For each data structure, the number of entries in the data structure.

This performance attribute is associated with each data structure in the program.

<u>Performance Attribute 10</u>: For each data structure, the service time of an entry in the data structure, i.e. the relative time that an entry remains in the data structure before being serviced.

This performance attribute is associated with each data structure in the program.

<u>Performance Attribute 11</u>: The rate of input to the module.

This performance attribute is associated with each module in the program involved in a PDR via the sensitivity to the rate of input mechanism.

<u>Performance Attribute 12</u>: For each dependent iterative structure in the module, the number of iterations.

This performance attribute is associated with critical sections of type seven.

Two additional performance attributes will be defined in Section 5.1.5 discussing the decomposition of performance requirements into performance attributes. These performance attributes are identified in the program in that step and are then added to the performance attribute data base.

5.1.4 Lexical Analysis Step 4

In this section all of the virtual performance attributes in the program will be identified. A data base which associates virtual performance attributes with modules and particular statements within the modules will be created. Each virtual performance attribute will now be defined and a description of how it is identified presented.

Virtual Performance Attribute 1: The execution time of the critical sections of a competing module which must wait due to denial of a particular contended resource.

This virtual performance attribute is associated with each module involved in a PIR via the shared resources mechanism and each statement in the module which requests the resource in contention.

Virtual Performance Attribute 2: The execution time of the critical sections of a module which contain an invocation to a module or the utilization of an abstraction.

This virtual performance attribute is associated with each dominant module involved in a PDR via the called modules mechanism or the abstraction mechanism and each statement in the module which invokes another module or an abstraction.

Virtual Performance Attribute 3: The execution time of the critical sections of a module which contain a dependent iterative structure.

This virtual performance attribute is associated with each dependent iterative structure in the module.

Virtual Performance Attribute 4: The execution time of the critical sections of code which contain a storage or retrieval request of an entry from a data structure.

This virtual performance attribute is associated with each module in the program utilizing a shared data structure and each statement in the module which references the shared data structure.

Virtual Performance Attribute 5: The execution time of the critical sections of code which must wait for a message to be transmitted from another module.

This virtual performance attribute is associated with each dependent module involved in a PDR via the interprocess communication mechanism and each statement in the module corresponding to a WAIT for the message. A module Y is the dependent module involved in a PDR with module X, if a change in module X can affect module Y via the PDR.

5.1.5 Lexical Analysis Step 5

In this section, the decomposition of performance requirements into performance attributes will be discussed. The decomposition of a performance requirement quantitatively into the effect of its corresponding performance attributes is a very complex task which is not attempted in this technique. Instead, the decomposition is qualitative in nature, i.e. performance attributes are identified which contribute to the preservation or violation of performance requirements without consideration of their relative magnitude towards the performance requirements. This simplification is justified because this maintenance technique attempts to identify performance requirements which may be violated due to the maintenance effort, and does not attempt to analytically confirm whether or not a performance requirement is actually violated. The guidelines for the decomposition of performance requirements into performance attributes are based upon the assumption that the performance requirements are testable and stated in terms of processing flows. A data base will be created containing the decomposition information. The data base will be utilized in later phases where the traceability of the decomposition of the performance requirements into the performance attributes is required.

The decomposition of the performance requirements into the performance attributes can be accomplished in the following way:

Step 1: Analyze each performance requirement. If the performance requirement states that the program executes in a specified time between two points, then identify the modules whose execution is governed by the performance requirement. The performance requirement is then decomposed into performance attributes of type six for each of these modules. If the module does not already have a performance attribute of type six, one is added to the performance attribute data base along with an appropriate critical section of type five.

If only a particular segment of a module is governed by the performance requirement, then the requirement is decomposed into a performance attribute of type 6'. Performance attribute 6' corresponds to the execution time of a segment of code. Critical section 5' is the corresponding critical section for performance attribute 6'. Both must then be added to the performance attribute and critical section data bases.

Step 2: If the performance requirement states that the program is executed with resource utilization restrictions between two points, then identify the modules whose execution is governed by the performance requirement. The performance requirement is then decomposed into performance attributes of type 7 for each of these modules.

If only a particular segment of a module is governed by the performance requirement, then the requirement is decomposed into a performance attribute of type 7'. Performance attribute 7' corresponds to the resource utilization of a segment of code. Critical section 7' is the corresponding critical section for performance attribute 7'. Both critical section 7' and performance attribute 7' must then be added to the performance attribute and critical section data bases.

5.2 Description of Ripple Effect Tracing Steps

In this section, a description will be presented for each of the performance ripple effect analysis tracing steps outlined in Section 4.

5.2.1 Tracing Step 1

In this step, the change management system (CMS) data base and the characterization of the program produced during lexical analysis are utilized to determine the set of program blocks involved in the maintenance change.

5.2.2 Tracing Step 2

In this step, all of the critical sections affected by the maintenance activity are identified. This is accomplished by utilizing the critical section data base created during lexical analysis to determine which critical sections a particular block is an element of. Thus, once the blocks undergoing maintenance are identified, the critical section data base can be referenced to determine which critical sections are affected by the maintenance activity.

This process can be completely automated through the utilization of the change management system. This system automatically keeps track of program changes during the maintenance activity and records them in the change management data base. After lexical analysis is complete, a simple procedure could be developed to map the program changes recorded in the change management data base into changes affecting program blocks, which in turn can be used to identify the critical sections by referencing the critical section data base.

5.2.3 Tracing Step 3

In this step, the performance attributes corresponding to the critical sections affected by the modification are identified. This identification is trivial since the critical sections and corresponding performance attributes were identified during lexical analysis and this information was recorded in the performance attribute data base. Thus, this step consists of only references to the performance attribute data base to identify the performance attributes associated with the critical sections affected by the maintenance activity.

5.2.4 Tracing Step 4

In this section, a description of how to identify all of the performance attributes in the program affected directly by the modification and by performance ripple effect will be presented. This requires an identification of the performance dependency relationships in existence among the performance attributes and virtual performance attributes. A performance dependency relationship is defined to exist between two performance attributes or between a performance attribute and a virtual performance attribute if a change in the performance attribute may affect the other performance or virtual performance attribute. If a virtual performance attribute is affected, the corresponding performance attributes associated with it must be identified.

A four step algorithm will now be outlined for identifying all of the performance attributes in the program affected by performance ripple effect. The input to the algorithm consists of a set X which contains the performance attributes directly affected by the modification. At termination of the algorithm, the set X contains all of the performance attributes affected directly by the modification or by performance ripple effect.

Step 1: Select a performance attribute, x, in X which has not been selected before. If there are no any new elements in X to be selected, then terminate.
Step 2: Utilizing the performance dependency relationships in existence among the performance attributes in the program, identify all the performance attributes in the program which may be affected by a modification of performance attribute x. Add these new performance attributes to X if they have not already appeared.
Step 3: Utilizing the performance dependency relationships in existence between the performance attributes and the virtual performance attributes in the program, identify all of the virtual performance attributes which may be affected by a modification of performance attribute x. If x does not affect any virtual performance attribute, go to Step 1, otherwise go to Step 4.
Step 4: For each virtual performance attribute identified in Step 3, determine the corresponding performance attributes associated with it. Add these performance attributes to X if they have not already appeared. Go to Step 1.

In the following sections, a description of each of the steps in this algorithm will be provided. In order to simplify our description, we will label the various mechanisms for propagation of performance changes as shown in Table 1.

5.2.4.1 Description of Processing Step 1

In this step, a performance attribute x is selected from X. The only criterion for selecting x is that it has not been selected before. Thus, some scheme must be implemented for keeping track of which performance attributes have been selected. Once all performance attributes have been selected, the algorithm terminates.

5.2.4.2 Description of Processing Step 2

In this step, all of the performance attributes in the program which may be affected by a modification of performance attribute x are identified by utilizing the performance dependency relationships in existence among the performance attributes in the program. These new performance attributes are added to X if they have not already appeared. The performance dependency relationships among the performance attributes in the program can be described according to a set of rules.

TABLE 1. LABELS FOR MECHANISMS FOR PROPAGATION OF PERFORMANCE CHANGES

Mechanism One	Parallel Execution
Mechanism Two	Shared Resources
Mechanism Three	Interprocess Communication
Mechanism Four	Called Modules
Mechanism Five	Shared Data Structures
Mechanism Six	Sensitivity to Rate of Input
Mechanism Seven	Execution Priorities
Mechanism Eight	Abstraction

The rules are of the format

MODULE X/PA.Y → MODULE Z/PA.W   CONDITION

and are interpreted as follows: A change in performance attribute Y of module X may affect performance attribute W of module Z if the condition is satisfied. A variation of this format is the replacement of MODULE with DS. which represents data structure. DS. X/PA.Y. is then interpreted as the Yth performance attribute of data structure X.

The rules for describing performance dependency relationships among the performance attributes in the program are the following:

MODULE X/PA.1 → MODULE Y/PA.1 if X is the dependent module involved in a PDR with module Y via mechanism one, four, or eight.

MODULE X/PA.1 → MODULE Y/PA.4 if X is involved in a PDR with module Y via mechanism one.

MODULE X/PA.1 → MODULE Y/PA.6 if X is the dominant module involved in a PDR with module Y via mechanism four, eight, or one.

MODULE X/PA.2 for resource i → MODULE Y/PA.2 for resource i if module X is involved in a PIR with module Y via mechanism two.

MODULE X/PA.2 for resource i → MODULE Y/PA.3 for resource i if module X = module Y or module X is involved in a PIR with module Y via mechanism two.

MODULE X/PA.2 for resource i → MODULE Y/PA.4 if module X is involved in a PIR with module Y via mechanism two.

MODULE X/PA.2 for resource i → MODULE Y/PA.7 for resource i if module X = module Y.

MODULE X/PA.3 for resource i → MODULE Y/PA.2 for resource i if module X is involved in a PIR with module Y via mechanism two.

MODULE X/PA.3 for resource i → MODULE Y/PA.3 for resource i if module X is involved in a PIR with module Y via mechanism two.

MODULE X/PA.3 for resource i → MODULE Y/PA.4 if module X is involved in a PIR with module Y via mechanism two.

MODULE X/PA.3 for resource i → MODULE Y/PA.7 for resource i if module X = module Y.

MODULE X/PA.4 → MODULE Y/PA.2 for resource i if module X = module Y.

MODULE X/PA.4 → MODULE Y/PA.3 for resource i if module X = module Y.

MODULE X/PA.4 → MODULE Y/PA.4 if there exists a module Z such that a PDR is in existence between module Y and module Z via mechanism two or three and module X has precedence over module Y.

MODULE X/PA.4 → MODULE Y/PA.5 for message i if module X = module Y.

MODULE X/PA.6 → MODULE Y/PA.4 if there exists a module Z such that a PDR is in existence between module Y and module Z via mechanism two or three and module X has precedence over module Y.

MODULE X/PA.6 → MODULE Y/PA.6 if module X is the dominant module involved in a PDR with module Y via mechanism four, seven, or eight.

MODULE X/PA.7 for resource i → MODULE Y/PA.7 for resource i if module X is the dominant module involved in a PDR with module Y via mechanism four or eight.

MODULE X/PA.7 for resource i → MODULE Y-ALPHA Z/PA.7 for resource i if module X is the dominant module involved in a PDR with module Y via mechanism four and the call to module X is in Alpha Z.

DS.X/PA.9 → DS.Y/PA.8 if DS.X = DS.Y

DS.X/PA.9 → DS.Y/PA.10 if DS.X = DS.Y

MODULE X/PA.11 → DS.Y/PA.9 if module X stores the input in data structure Y.

These rules can be utilized to determine exactly which performance attributes in the program may be affected by a modification of performance attribute x. The appropriate rules for performance attribute x are first identified. A check is then made to determine if the condition for the performance dependency relationship rule is satisfied. This determination can be made by examining the performance dependency relationship data base created during identification of the mechanisms in lexical analysis. If the condition is satisfied, then the performance attribute affected by the rule is added to set X if it has not already existed in X.

5.2.4.3 Description of Processing Step 3

In this step, all of the virtual performance attributes in the program which may be affected by a modification of performance attribute x are identified by utilizing the performance dependency relationships in existence from the performance attributes to the virtual performance attributes in the program. The performance dependency relationships from the performance attributes to the virtual performance attributes can be described according to a set of rules. The rules are of the format

MODULE X/PA.Y → MODULE Z/VPA.W   CONDITION

and are interpreted as follows: A change in performance attribute Y of module X may affect virtual performance attribute W of module Z if the condition is satisfied. The rules for describing performance dependency relationships between the performance attributes and the virtual performance attributes in the program are the following:

MODULE X/PA.2 for resource i → MODULE Y/VPA.1 for the request for resource i if module X is involved in a PIR with module Y via mechanism two.

MODULE X/PA.3 for resource i → MODULE Y/VPA.1 for the request for resource i if module X is involved in a PIR with module Y via mechanism two.

MODULE X/PA.6 → MODULE Y/VPA.2 for each statement invoking module X in module Y.

DS.X/PA.8 → MODULE Y/VPA.4 for each statement referencing data structure X in module Y.

MODULE X/PA.5 for message i → MODULE Y/VPA.5 for the statement corresponding to a WAIT for message i.

MODULE X/PA.12 for dependent iterative structure i → MODULE Y/VPA.3 for dependent iterative structure i if module X = module Y.

These rules can be utilized to determine exactly which virtual performance attributes in the program may be affected by a modification of performance attribute x. The appropriate rules for performance attribute x are first identified. A check is then made to determine if the condition for the performance dependency rule is satisfied. This determination can be made by examining the performance dependency relationship data base created during the identification of the mechanisms in lexical analysis. If the condition is satisfied, then the virtual performance attribute affected by the rule is identified and undergoes further processing in Step 4.

5.2.4.4 Description of Processing Step 4

In Step 3 of the algorithm the virtual performance attributes affected as a consequence of modification of performance attribute x were identified. For each virtual performance attribute, a set of performance attributes can be associated with it such that if the virtual performance attribute is affected, the performance attributes are also affected. In this step, the performance attributes associated in this way with a particular virtual performance attribute must be identified. This identification is dependent of the particular type of virtual performance attribute which is being considered. In the following sections simple algorithms will be presented for identifying the performance attributes associated with each of the virtual performance attribute types. Thus, for each of the virtual performance attributes identified in Step 3, the appropriate algorithm can be executed to identify the performance attributes associated with it. These performance attributes are then added to set X if they have not already appeared.

5.2.4.4.1 Algorithm for Identifying Virtual Performance Attributes of Type One

Step 1: Utilizing the critical section data base, identify the critical sections of the module which contain the request for the resource in contention associated with the virtual performance attribute of type one.
Step 2: Utilizing the performance attribute data base, identify the performance attributes of the module which are associated with these critical sections. These performance attributes are then associated with the virtual performance attribute.

5.2.4.4.2 Algorithm for Identifying Virtual Performance Attributes of Type Two

Step 1: Utilizing the critical section data base, identify the critical sections of the module which contain the statement which invokes the module or abstraction which is associated with the virtual performance attribute of type two.
Step 2: Utilizing the performance attribute data base, identify the performance attributes of the module which are associated with these critical sections. These performance attributes are then associated with the virtual performance attribute.

5.2.4.4.3 Algorithm for Identifying Virtual Performance Attributes of Type Three

Step 1: Utilizing the critical section data base, identify the critical sections of the module which contain the iterative code controlled by the dependent iterative structure which is associated with the virtual performance attribute of type three.

Step 2: Utilizing the performance attribute data base, identify the performance attributes of the module which are associated with these critical sections. These performance attributes are then associated with the virtual performance attribute.

5.2.4.4.4 Algorithm for Identifying Virtual Performance Attributes of Type Four

Step 1: Utilizing the critical section data base, identify the critical sections of the module which contain the reference to the data structure which is associated with the virtual performance attribute of type four.

Step 2: Utilizing the performance attribute data base, identify the performance attributes of the module which are associated with these critical sections. These performance attributes are then associated with the virtual performance attribute.

5.2.4.4.5 Algorithm for Identifying Virtual Performance Attributes of Type Five

Step 1: Utilizing the critical section data base, identify the critical sections of the module which contain the WAIT statement for the message associated with the virtual performance attribute of type five.

Step 2: Utilizing the performance attribute data base, identify the performance attributes of the module which are associated with these critical sections. These performance attributes are then associated with the virtual performance attribute.

5.2.5 Tracing Step 5

In this step, all of the performance requirements which are affected by a change in any of the performance attributes involved directly with the modification or through performance ripple effect are identified. All of the performance attributes involved directly with the modification or through performance ripple effect have previously been identified and are elements of set X. The corresponding performance requirements affected by changes in these performance attributes can then be identified by utilizing the data base containing the traceability of the decomposition of the performance requirements into the performance attributes performed during lexical analysis.

6.0 Integration of the Performance Ripple Effect Analysis and the Logical Ripple Effect Analysis Processing Steps

In Part I of this handbook a logical ripple effect analysis technique to improve the maintenance process was described. In Part II of this handbook a performance ripple effect analysis technique to improve the maintenance process was also described. The performance ripple effect analysis technique should be used in conjunction with the logical ripple effect analysis technique. In this section, the required processing steps of the performance ripple effect analysis technique will be integrated with those of the logical ripple effect analysis technique in order to produce a unified ripple effect analysis technique. The unified ripple effect analysis technique will consist of a lexical analysis phase and a tracing phase.

6.1 Lexical Analysis Phase

The first phase of the ripple effect analysis technique is the lexical analysis phase. In this phase, the program is analyzed with respect to the proposed modification and a characterization of the program is compiled and saved in a data base. The characterization of the program contains the information necessary for tracing both logical and performance ripple effect. A description of the required processing steps involved with lexical analysis will now be presented.

Step 1: Perform the Text-Level Lexical Analysis to produce a program graph based on program blocks, compute the error flow properties of each program block, and construct the invocation graph.

Step 2: Perform the System-Level Lexical Analysis to derive the precedence ordering among modules, compute the module error characteristics sets, and update the block error characteristic sets.

Step 3: Identify all the mechanisms for the propagation of performance changes and the corresponding performance dependency relationships in the program.

Step 4: Identify all the critical sections in the program in terms of the program blocks identified in Step 1.

Step 5: Identify all the performance attributes in the program.

Step 6: Identify all the virtual performance attributes in the program.

Step 7: Decompose the performance requirements for the program into the performance attributes which contribute to the preservation or violation of the performance requirements.

6.2 Tracing Phase

The second phase of the ripple effect analysis technique consists of tracing the logical and performance changes, i.e. the ripple effect which occurs as a consequence of the maintenance changes. The input to the technique in this phase includes all of the information about the program collected and stored in a data base during the lexical analysis phase. A description of the required processing steps involved with the tracing phase will now be presented.

Step 1: Utilizing the change management system data base and the characterization of the program produced during lexical analysis, identify the set of blocks and their primary error sources initially involved in the change for each module in the program.

Step 2: Based upon the blocks involved in the change identified in the previous step and the characterization of the program produced during lexical analysis, identify all of the critical sections affected by the maintenance activity.

Step 3: For each of the critical sections affected by the modification, determine the corresponding performance attributes which may be affected if the critical section is modified. The correspondence between performance attributes and critical sections was generated during lexical analysis.

Step 4: Form a set $\bar{\mathcal{M}}$ composed of modules initially involved in the change.

Step 5: Compute the error flow of set $\bar{\mathcal{M}}$. Let the set of modules affected by the error flow define set $\mathcal{M}^*$ and the set of blocks and their error sources within each module M_j which contributes to error flow define L_j^*.

Step 6: Apply the ripple effect criterion to each element in $\mathcal{M}^*$. Let all modules in $\mathcal{M}^*$, which require additional maintenance due to the ripple effect criterion, define set $\mathcal{M}^R$.

Step 7: Apply the block elimination criterion to each element in $\mathcal{M}^*$. Let all blocks and their error sources within M_j which require additional maintenance activity define L_j^R. The maintenance personnel must check all of the blocks in L_j^R for each module in $\mathcal{M}^R$ to insure that they are consistent with the initial change.

Step 8: Trace the performance ripple effect among the performance attributes utilizing the performance dependency relationship rules in order to identify all of the performance attributes affected by the modification.

Step 9: Identify the performance requirements which are affected by a change in any of the performance attributes involved directly with the modification or through ripple effect. These performance requirements can be identified by the traceability of the decomposition of the performance requirements into the performance attributes performed during lexical analysis.

In Section 6.1 and 6.2 the required processing steps of the unified ripple effect analysis technique were presented. The complete description of each of these steps can be found in Section 5 of Part I and Part II of the handbook.

7.0 References

[1] Yau, S. S., Collofello, J. S. and Hsieh, C. C., "Self-Metric Software Vol II"--A Handbook: Part I, Logical Ripple Effect Analysis, Final Technical Report.

[2] Yau, S. S., Collofello, J. S., and MacGregor, T., "Ripple Effect Analysis of Software Maintenance," Proc. of COMPSAC 78, pp. 60-65.

[3] Yau, S. S. and Collofello, J. S., "Performance Considerations in the Maintenance Phase of Large-Scale Systems," RADC-TR-79-129, June 1979.

[4] Yau, S. S. and Collofello, J. S., "Performance Ripple Effect Analysis for Large-Scale Software Maintenance", RADC-TR-80-55, December, 1979.

[5] Yau, S. S., "Self-Metric Software--Summary of Technical Progress", Vol I Final Technical Report.

[6] Alford, M. W., "A Requirement Engineering Methodology for Real-Time Processing Requirements," IEEE Trans. on Software Engineering, Vol. SE-3, January 1977, pp. 60-69.

[7] Gannon, C., Brooks, N. B. and Urban, R. J., "JAVS Technical Report, User's Guide," RADC-TR-77-126, Vol. I, 1977.